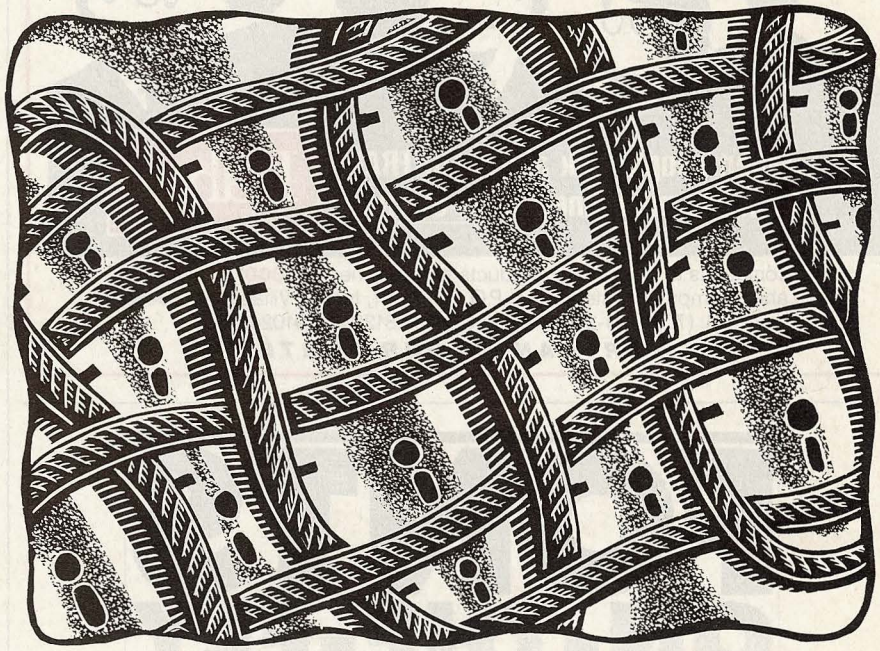
OS/2 NOTEBOOK ■ Douglas A. Hamilton

MASTERING OS/2 THREADS

OS/2's unique style of lightweight multitasking taxes developers and rewards users



Don't make me wait! When I select Print, insert a new value into a spreadsheet, or resize a desktop publishing window, I don't want to be ignored while the application grinds away. None of us likes to wait, and applications designers have heard that message.

So spreadsheets today come with background recalc features, for example. But these features have taken forever to arrive, and they still aren't universally provided. If everyone knows what the problem is, why don't they just go ahead and fix it?

The answer is that background anything under DOS or Windows has to be custom-made. OS/2 is the first and only widely distributed system to provide a specific mechanism for dealing with problems of this class without building everything from scratch. OS/2 lets a designer cleave off compute- or I/O-bound activities as separately scheduled threads and thereby ensure crisp user interaction at all times.

Modes of Concurrency

When I talk about threads, I'm talking about concurrency: doing more than one thing at a time. The whole idea is to avoid having a processor sit idle when it could be doing something useful, and to be sure that what it is doing is most important.

Key factors that determine the performance of a concurrency mechanism are the time required to create and switch between tasks and the ease with which tasks can share information. Because threads carry less state information than normal OS/2 or Unix processes, the system can create and switch among them quickly. Because they share memory, tasks enjoy high-bandwidth communication.

The idea isn't completely new. Researchers in the Unix community, particularly at the Carnegie Mellon University Mach project, have talked about lightweight processes for several years. But OS/2 is the first commonly available system to implement this strategy.

A thread is a simple flow of control within a process. Its state consists of an instruction pointer, a stack, a register set, its priority, and certain types of semaphores. Everything else—memory (i.e., instructions and data), file descriptors, even the current disk and directory—is shared with the other threads in the process. Threads, like interrupt routines, require the designer to identify critical sections and implement resource-sharing protocols.

Threads run inside processes, which in turn run inside screens. The progression from threads outward to screens entails more and more "fire-walling" on the part of OS/2.

But the most important distinction is that while processes and screens are normally used for sharing the processor between applications, threads are uniquely a way of sharing the processor inside an application. That means more responsive single-user applications and, just as important, high-performance server applications. Distributed databases that manage transactions using threads, rather than entire processes, can be highly efficient.

Where Are the Applications?

So now I'm back to almost the same question: If everyone knows what the problem is, and if OS/2 provides the means of solving it, why don't you see

continued

OS/2 NOTEBOOK

many great multithreaded OS/2 applications?

To start, building a multithreaded application takes a tremendous amount of hard work. Elaborate handshaking is required to ensure that threads don't trample over each other. Everything has to be reentrant, compiled with the right options, and linked with the right libraries. Any shared resources have to be semaphored, and all that semaphoring has to be carefully constructed to avoid race conditions anywhere that could result in "deadly embrace." If the term deadly embrace is a bit fuzzy, trust me, writing your first multithreaded application will give you a good visceral feel for it.

Whenever a thread needs to "own" something, you have to invent a mechanism for the purpose. For example, when building the Hamilton C shell, a highly multithreaded command processor for OS/2, I had to come up with a way for a thread to maintain the notion of a current directory. It would hardly have been acceptable if a script running quietly in the background could suddenly, without warning, change the foreground current directory. Building a high-performance mechanism to re-create a current directory notion for each thread turned out to be a challenging project.

Debugging can be a real treat. Since the kernel's decisions about what thread gets to run next depend on what segments are loaded, setting a breakpoint can (by forcing a segment to be loaded) cause a different execution order. Here's the software analog of the hardware bug that disappears when you put the scope probe on it.

Not for the Faint of Heart

When I began working on the C shell in the summer of 1987, I worried a lot about possible competitors doing the same thing (i.e., building Unix-style tools for OS/2). It seemed like an obvious need, and I knew others were equally capable of writing such things. But mostly, that didn't happen. I wondered why.

One thing that I suspect is that most people who did try to build OS/2 applications came from the DOS world. Swamped by the sea change to multitasking and multithreading, they had difficulty making headway. Time invested with DOS, unless it was spent working on device drivers (which raise acute issues of concurrency), isn't good training for OS/2 threads.

Documentation didn't help. I remember opening my first OS/2 Software Development Kit (SDK) and reading that "a

continued on page 110

thread is a dispatchable element used by MS OS/2 to track execution cycles of the processor." Three years later, that still doesn't tell me anything.

Then came the infamous MTDYNA .DOC. At first, you couldn't use the C library if you wanted to use threads, because the library routines weren't reentrant. In the spring of 1988, a new release of the C compiler brought the multithreaded library and headers and a 1039-line read-me file, MTDYNA.DOC, buried on one of the disks. For two years, that was the only official documentation for most of us.

Other roadblocks have been the constantly changing "musical header files." I know I've not been alone in just dreading each new SDK or Toolkit release. Each one seemed to bring a new set of seemingly gratuitous changes to all the names defined in the headers. First it was all uppercase, then mixed case; first English, then Hungarian. Each release meant nothing would compile until I'd made all the same gratuitous changes to all my own source code.

Least Common Denominator

The other big reason that there aren't many great multithreaded applications is that once you write one, it's not portable. Conversely, if you're porting something in from another environment, you don't just add threads and stir. To really use threads, you have to weave them pretty tightly into the fabric of your product. And let's face it: It's one thing to be nonportable if you're selling to an installed base of 50 million users, and quite another to a base of only 300,000.

Not surprisingly, most of the first wave of applications for OS/2 have been ports from DOS or Windows. Microsoft's own Word 5.0 and Excel are two very disappointing but typical examples of programs that do absolutely nothing to take advantage of OS/2. Neither Word nor Excel will do background printing; Excel doesn't even let you move its window around while it prints.

The arrival of Windows 3.0 clouds things further. With the upcoming capability of OS/2 2.0 to run Windows binaries unmodified, many developers may think that the right answer is the purely opportunistic one: Write it for Windows, and if it works on OS/2, fine, but don't do anything special. In other words, don't use threads.

What's the Prognosis?

In my view, the prognosis is mixed. On the technical side, things have improved. Documentation is much better. Many books show how to write a multithreaded program. From discussions I see on BIX and elsewhere, most developers seem to be gaining the familiarity and experience they need.

OS/2 2.0 promises a new, improved semaphore application programming interface that's touted as easier to use, although I'm skeptical. In my experience, it's not the semaphore primitives that are at fault, it's that semaphores are inherently tricky. Race conditions are just plain tough to avoid and even tougher to debug.

I see nothing that changes that. Some version 2.0 changes appear to be more musical headers. For example, the so-called FS (fast, safe) semaphores introduced with great fanfare last year are gone. What possible reason could there have been to introduce these semaphores at all if they were going to be eliminated so quickly?

But the biggest impediments to seeing all those great multithreaded applications now and through the rest of the year will be nontechnical. If sales of OS/2 continue at current levels, don't expect much.

Still, there's hope. Although Windows 3.0 will likely give all of us in the OS/2 community gas pains, it may ultimately be the best thing that could happen to OS/2. If you have the hardware to run Windows 3.0 acceptably, OS/2 should run fine also. OS/2 2.0 will make the migration easier. I have one DOS box right now, and, for me, it's one too many, considering how often I bother with it. But I admit even I was strangely captivated to see multiple DOS applications like good old Lotus 1-2-3 running in Presentation Manager windows under OS/2 2.0.

Ultimately, competitive pressures will grow as more users and developers learn just what can be done with threads. Unlike breakfast cereal, where you can eat the whole box and still have no idea whether it's any good for you, most folks figure out pretty quickly whether new software is any good for them. Take heart: OS/2 threads, used properly, are very good for you—enough so anyone can notice. ■

Douglas A. Hamilton is the founder of Hamilton Laboratories in Wayland, Massachusetts, and the author of the Hamilton C shell, a command processor and utilities package for OS/2. He can be reached on BIX as "hamilton."

Your questions and comments are welcome. Write to: Editor, BYTE, One Phoenix Mill Lane, Peterborough, NH 03458.